AD-A243 165

DTIC
ELECTE
DEC 1 0 1991
S D
C

UNED

September 1991

MTR11209

Jonathan I. Leivent
Ann M. Wollrath

Resource Allocation in Distributed Memory Multiprocessing

Approved for public release;
distribution unlimited.

MITRE

Bedford, Massachusetts

91-17466

91 1209 118

September 1991

MTR11209

Jonathan I. Leivent
Ann M. Wollrath

Resource Allocation in Distributed Memory Multiprocessing

CONTRACT SPONSOR MITRE-Sponsored Research
CONTRACT NO. N/A
PROJECT NO. 9124A
DEPT. G117

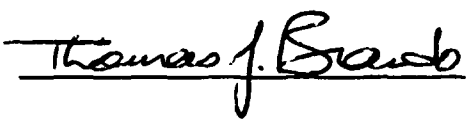
Approved for public release;
distribution unlimited.

Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

MITRE

The MITRE Corporation
Bedford, Massachusetts

Department Approval: 

MITRE Project Approval: 

ABSTRACT

The problems of load balancing and locality of reference for large distributed memory message-passing multiprocessors running object-oriented applications are studied. The object-oriented nature of the applications implies that tasks must be executed on the processor containing the host object for the task (tasks are method executions, the host object is the destination of the message that invokes the method). The particular constraint for the study is that objects will not be moved between processors once allocated. The primary resource allocation problem then becomes the determination of the best processor on which to allocate each new object during the execution of the application. The study uses discrete event simulation of pseudo-applications under the control of several resource allocation policies selected as limiting cases of practical solutions. Results are presented, and some general rules-of-thumb are suggested for use in real systems.

TABLE OF CONTENTS

SECTION 1

INTRODUCTION

One principal challenge of managing distributed memory multiprocessor systems is resource allocation. Resource allocation strategies attempt to reduce the execution time of programs by assigning computation to processors in order to balance processor load and reduce communication delays. Our goal is to use resource allocation strategies to reduce the total execution time of object-oriented applications on distributed memory multiprocessors, where total execution time is defined as the interval between application startup and termination. We do not attempt to reduce the impact of one application on a multiprocessor running many applications; we view the multiprocessor as a single-user machine running only one application at a time. Also, we are interested in the performance of programs that have a very large number of tasks, so we favor resource allocation strategies that perform best as the size of programs (number of tasks) increases.

Generally, it is intractable to find the optimal allocation of processes (tasks) to processors, even if the evolution of processes within the system can be analyzed statically. The dynamic behavior of programs further complicates any resource allocation strategy, since the evolution of processes in such programs is data dependent. However, suboptimal resource allocation strategies can still yield important performance benefits.

Such suboptimal strategies may actually yield performance benefits approaching optimal strategies asymptotically as the size of the systems in question (number of tasks, number of processors) increases. This is certainly true when minimizing the average task execution time on a multiprocessor used as a multiprogramming environment, where the central limit theorem of statistics implies that a random allocation of tasks to processors would perform well as the number of tasks increases (the total task execution load assigned per processor would approximate a normally distributed random variable with decreasing variance as the number of tasks increases). However, if performance is defined as the maximum processor execution time, then the central limit theorem does not promise that good results are so easily achieved.

In this section, we will detail the problem of resource allocation for distributed object-oriented programs and compare it to previous work on similar problems. Section two introduces our resource allocation algorithms. Section three describes our simulation technique, and section four presents the simulation results.

1.1 PROBLEM DESCRIPTION

As previously stated, our goal is to investigate the problem of managing resources for object-oriented applications executing on distributed-memory multiprocessors. Before describing

the problem of resource allocation in detail, we briefly review here the concepts of object orientation, distributed memory, and their compatibility.

1.1.1 What is Object-Oriented Programming?

A programmer using the object-oriented paradigm [7] conceptualizes a problem to be solved in terms of communicating active objects and their behaviors. The objects have mutable state, represented by instance variables. Objects' behaviors are represented by function-like methods. Objects communicate to other objects by sending messages. When an object receives a message, it invokes the corresponding method, which may change the object's state and/or send messages to other objects. There are several other concepts associated with object-oriented programming, including inheritance and late binding, which are not important to our study.

1.1.2 What is Distributed Memory?

Distributed memory multiprocessors are composed of individual homogeneous processor elements, each having local memory and corresponding address space. There is no global shared address space. Processor elements communicate with each other by transmitting messages through an interconnection network of some type.

1.1.3 Mapping OOP onto Distributed Memory Architectures

Data access within standard non-object-oriented procedural languages (e.g., Pascal, C, Ada, FORTRAN) consists of simple reads and writes. Within any one procedure, these reads and writes may be scattered throughout the address space.

The primary difference between object-oriented programming and non-object-oriented programming that we are exploiting is the object-oriented abstraction of data accesses above the level of simple reads and writes so that these accesses incorporate portions of the execution of the program. In fact, in a fully object-oriented program, there is no execution that is not contained within data accessing abstractions.

The data encapsulation property present within object-oriented programming languages facilitates the use of distributed memory hardware by creating an abstraction of data access that is sufficiently removed from direct addressing. Objects are addressed through identifiers, which can be mapped to any of several implementations (including direct addressing). The data internal to an object can be considered to exist in an address space unique to that object. All methods invoked on an object can be viewed as executing within the object's address space (so long as call-by-value semantics are used for argument passing). Thus, one or more objects can be allocated to each individual processor without any added semantic interference from the processors' separate address spaces.

1.2 COMPARISON TO PREVIOUS WORK

A more extensive overview of various techniques for handling distributed resource allocation can be found in [8, 19].

1.2.1 Task Model vs. Object Model

In our model, the object, together with all method invocations on that object's data, is the unit of work load that is allocated to a processor. Objects tend to be a long term commitment on the part of a processor, often persisting for a significant portion of the program's lifetime.

Furthermore, objects and messages in the object-oriented model are created during the execution of methods on other objects. Initially, there is only one object. The creation of the rest of the objects occurs during execution at a rate that depends on the rate of execution, and hence on system load. This scheme differs considerably from the task model, in which tasks are assumed to be injected into the system from some outside source at a rate independent of the load on the system.

Much of the work in the area of dynamic resource allocation is targeted toward allocating independent tasks to processors [6, 13]. In these models, the unit of work is a task. The arrival rate of tasks to the system is generally independent of the state of the system, and is often viewed as constant. The goal of such tasking models is generally to increase the throughput of tasks. Systems such as these can be represented adequately by queuing models.

The resource allocation problem that we have studied is not directly amenable to analysis using standard queuing models. Primarily, this is due to the variation in object creation rates over time resulting from differing levels of parallelism. We have looked into designing an appropriate Markov process model, but have not made progress worthy of mention here.

Other research in resource allocation deals with shared memory multiprocessors or emulates a shared memory multiprocessor by providing virtual shared memory [18].

1.2.2 Static and Dynamic Analysis Issues

Others advocate the use of static analysis to resolve the problem of resource allocation in distributed systems [10, 20]. If the characteristics of the program are known beforehand, then static analysis of the program may help to determine the appropriate placement of tasks among processors in the system. Information such as an estimate of how long each task takes to run and the relationship between tasks (i.e., how often they communicate), if available, is valuable in determining where to allocate resources. For example, it may be advantageous to place tasks which communicate frequently on the same processor (in local memory) since this minimizes communication distance to a local reference. Alternatively, tasks communicating less frequently can be placed on different processors. Some static analysis

techniques can detect concurrent computations (threads) that should not be placed on the same processor.

There are several problems with static analysis that make its use prohibitive to us as a basis for resource allocation decisions. In our model, program characteristics are not known in advance. The behavior of many programs is largely data dependent and cannot be characterized ahead of time. The style of programming used for object-oriented methodologies makes static analysis even less attractive due to the frequency of dynamic data dependencies. Even if analysis makes use of programmer annotations about program behavior, static analysis techniques can, at best, predict only "typical" program behavior.

The resource allocation problem becomes much easier if program structure is known in advance. For example, a load balancing algorithm for applications which demonstrate periodic behavior [9] and a resource allocation strategy for programs with data parallel computations [15] have been proposed.

Dynamic program behavior can be determined using run-time analysis techniques. During program execution, data can be collected about processor utilization, memory utilization, and/or data dependencies. This information can be coordinated by a control mechanism (centralized or distributed) to make more informed resource allocation decisions possible. The question with run-time analysis of program behavior is: once this information is collected, how is it best used to make resource allocation decisions? Also, program behavior is constantly changing, so some out-of-date information must be contended with. Another factor is the overhead associated with any run-time data gathering technique; the overall gain in system performance must outweigh the overhead incurred by gathering, analyzing, and acting on program characteristics determined at run-time.

[14] proposes a dynamic resource allocation scheme where both system architecture and program structure must be hierarchical (i.e., hierarchical multiprocessor and task structure). The program must also have the capability of adjusting grain size dynamically.

Our interest in distributed object-oriented programs stems from other research which is currently investigating the application of the Time Warp simulation paradigm to general purpose object-oriented computation [1, 2, 17]. Any Time Warp [11] based run-time system is sensitive to message arrival order. Under different orderings of message arrivals, an object will execute different computations. Despite the fact that all object states will later be brought into agreement with what would be achieved in a sequential execution, the excess computation obviously does have an impact on resources, especially on processor load. Therefore, accurate time predictions of the sequential execution of a program, or even of parallel executions with differing message arrival times, would not accurately predict processor load in any given parallel execution.

1.2.3 Moving Objects

Others advocate periodic remapping of tasks to processors during execution [10, 12, 15, 16]. Because the target run-time system we have in mind is based on Time Warp, there is a considerable memory overhead per object moved. This phenomenon would make the cost of relocating objects prohibitively large. As a result, we do not consider relocating objects to balance load or increase locality of reference.

1.2.4 Locality of Reference

Originally, our study investigated the importance of locality of reference in distributed resource allocation. However, our tests showed that load balancing considerations far outweigh locality of reference considerations for the applications tested. The reasoning behind this is explained below. Others have investigated the impact of locality of reference specifically in shared memory non-uniform memory access (NUMA) machines [4, 5].

SECTION 2

RESOURCE ALLOCATION ALGORITHMS

This project began without a clear picture of which algorithms to test. There were several initial algorithms that were far more complex than the ones included in the final study. The difficulties of dealing with the complexity of the initial algorithms lead us to develop simplified cases that could be used to study individual concepts. The simplified algorithms represent ideal limiting cases where many aspects of resource allocation algorithms are completely ignored. For example, in most of the tests, message latency was eliminated. In other tests, some interprocessor communication was done without messages at all. In all cases, computation involved in the resource allocation procedure internal to a processor incurred no simulation time overhead.

The results of each simplified test pointed us towards other algorithms, which were tested in turn. This iterative procedure continued until we found a particular class of algorithms that behave nearly as well as our ideal baseline case, but have practical implementations.

In this section, we present the components of the tested algorithms and apply them to several cases. We will describe the case in which every processor has perfect knowledge of the state of all processors. This "omniscient" algorithm is used as a baseline for our study. Next, we relax the amount of information each processor requires about other processors and describe several classes of algorithms developed using the relaxed criteria. At the other end of the spectrum, a random allocation algorithm is used as the second baseline for our study. The goal is to find resource allocation algorithms which approach the results of the omniscient algorithm, but perform no worse than the simplistic random allocation algorithm which uses no information about processor load.

2.1 ALGORITHM COMPONENTS

The resource allocation algorithms discussed here can each be divided into three components: a processor *load metric*, a processor *choice method*, and a *source-destination handshaking protocol*. A processor load metric is used to determine which processors are "good" candidates for allocation. A processor choice method deals with how a destination processor is selected once the load metric for each processor is known. The source-destination protocol determines the respective duties of each processor (source and destination).

2.1.1 Processor Load Metrics

Each resource allocation algorithm contains a method for estimating the relative future execution loads on processors by examining the processors' current state. The ideal resource

allocation algorithm would be able to accurately determine the future load on every processor.

Future Load. The actual future load of each object, a measure of the computation time required to execute all yet-to-be-received messages for the object, is an unrealistic metric to use for practical resource allocation algorithms. However, for the purposes of our study, this ideal metric can be used as a baseline for comparison with other metrics. The purpose of such an ideal baseline is to determine when possible additions to algorithms cannot increase performance sufficiently to merit the additional effort and complexity.

We can include the ideal metric in our study because we drive each simulation from an execution trace which contains sufficient information about future loads. Note that an algorithm that uses the ideal metric is not necessarily optimal. An optimal algorithm must use information about future object creations in conjunction with the ideal metric to determine optimal object placement. This information is present in the execution traces that drive the simulations; however, we do not attempt to simulate the optimal algorithm because of its intractability.

Number of Objects. When future load information is not available, as would be the case in realistic systems, the resource allocation algorithm must use some alternative metric that can be expected to order processors in approximately the same way as future load measures would. One piece of information readily available on all processors is the number of objects allocated to each. The future load on a processor is expected to be very roughly proportional to the number of objects. This is especially true if some garbage collection algorithm removes objects from processors once it can establish that the objects will have no future tasks to perform. Using the number of objects to predict future load would not be expected to work well if objects tend to remain inactive for long periods of time, or if some objects require much more processor time than others.

Memory Used. When all objects are nearly the same size (in terms of memory consumed), then the number-of-objects metric is nearly equivalent to a metric that takes into account the amount of memory consumed on the processor. However, if the execution time requirements of an object are somewhat proportional to the size of the object, then a metric based on the amount of memory used would perform better than one based on the number of objects. Conversely, if the execution load of an object is not correlated to the object's size, then the amount-of-memory metric may not be beneficial.

Message Queue Length. As messages arrive at a processor, they are queued prior to being delivered to their destination objects. The length of a processor's message queue can thus be used to gauge the load on the processor in the near future. Using the time-weighted average of the message queue length as the resource allocation algorithm's load metric works particularly well if the execution times of messages do not vary too greatly.

Average Message Queue Wait Time. Statistics of a processor's previous load can be used to predict its future load to varying accuracy. One such statistic is the average time that a message spends on the processor's input queue. The average time on the queue would be expected to more closely parallel the actual load on the processor than would the queue length metric, since the average time does not assume that the execution times of messages are similar. However, average time on the queue as a metric is not necessarily a predictor of future load on the processor, as is the queue length.

Distance. Minimizing the actual communication distance between source and destination processors may be advantageous in cases where message latency contributes significant overhead to the running program. This metric may have little effect if locality of reference is not important. For a more complete discussion on the issues of locality of reference, see the discussion below on the non-uniform random baseline algorithm.

No Metric. The other baseline for our study is a resource allocation metric that does not attempt to predict future load at all: all processors are viewed by the algorithm as equal regardless of their state. We consider any algorithm that does not perform at least as well as this zero-information baseline case as having "failed" in a sense. We will address a common attribute of many of the failed algorithms that, when added to many successful algorithms, will cause them to fail (see the discussion of "piling on" in section 4).

2.1.2 Processor Choice Methods

Once a resource allocation algorithm has determined a ranking for each possible destination processor for a particular allocation using some metric, it must then pick the actual destination processor. The obvious approach would be to choose the highest ranking processor. However, our preliminary studies show that the highest ranking processor is not always the best choice. Therefore, we have developed several alternative methods of selection.

The first and most obvious selection strategy we studied is to select the best processor satisfying the metric. However, several processors may rank equally according to a single metric. Consequently, a tie breaking strategy needs to be employed. One tie breaking strategy would be to use a second processor load metric to differentiate among the "tied" processors. Another strategy would minimize the communication distance between source and destination processors. A third strategy is to select a processor arbitrarily from among the highest ranked processors.

Instead of consistently choosing the best processor, an alternative approach would select a processor from a set of processors not necessarily all ranked highest according to the processor load metric. The set is constructed in either of two ways. A collection of possible destination processors can be sorted by the load metric and a set constructed from some percentage of the top of the list. Alternatively, the set can be formulated by using a cutoff measurement that is equal to the average metric for all processors plus or minus some

number of standard deviations. The set can then be constructed from those processors whose metric is better than the cutoff; i.e., the worst processors are filtered out.

Once the set is formed, a destination processor can be selected in one of three ways: uniform random, non-uniform random, or round-robin. Both uniform random and round-robin selection strategies view each possible choice of destination processor as equivalent. The difference between uniform random and round-robin selection is that round-robin uses the history of previous choices by the source processor to determine the next choice. In effect, resource allocations are equally distributed among all destination processors from the point of view of the source processor. Non-uniform random attempts to minimize communication distance by favoring processors nearby. One aim of our investigation was to study the effects of a non-uniform random distribution on increasing locality of reference among objects.

2.1.3 Source-Destination Protocol

In any resource allocation algorithm, the processor load metric needs to be tested to determine the allocation destination, and updated when the destination is finally chosen. The *source-destination protocol* refers to which processor performs the test and which processor performs the update. However, to simplify our description of processor load metrics and choice methods, we have biased our discussion up to this point by suggesting that the source processor performs the metric test and destination selection tasks. In reality, the source or destination processor can do either task, resulting in significantly different allocation algorithms. For example, the source processor can choose the destination arbitrarily, and the destination can determine its suitability for allocation based on a processor load metric, and possibly refuse the allocation request. Furthermore, a processor load metric, such as number of objects, can either be updated when the object is allocated on the destination processor or updated immediately when a processor is chosen to be the destination. In some algorithms, allocation and update do not occur simultaneously because of message latency and queue delay.

None of the processor load metrics discussed above guarantees that the best possible destination processor, as determined using the metric, has sufficient memory and other resources to satisfy the allocation request. Thus, in addition to testing the processor load metric, either the source or destination processor must determine if sufficient resources are available at the destination. If the source processor does not determine whether sufficient resources are available at the chosen destination, the destination may refuse the allocation. In this case, control returns to the source processor, which must choose an alternative destination.

Once the destination for the allocation is determined, the destination processor must be updated to reflect the allocation. The update actually consists of three parts: reserving the necessary space for the object, updating the load metric on the destination processor, and initializing the object (permitting it to execute methods). The initialization of the allocated object takes place in all our tests when the allocation request message is received and

processed. The other two parts of the update can be done either atomically without sending messages, or when the allocation message is received and processed. The distinction between these two implementations turns out to be a very important factor in the performance of an allocation algorithm, and will be discussed later.

2.2 BASELINE ALGORITHMS

2.2.1 Random

Random allocation policies are non-adaptive policies, since the decision to place an object in a particular processor does not depend on system state (which is analogous to using no processor load metric). Some variations on random policies are discussed in [6, 13].

The baseline algorithm of the least complexity is one that uses a uniform random distribution to select the destination processor. The algorithm does not use state information of any processors in the system to make its decision. Also, the algorithm does not modify the state of the source processor, so successive allocations are independent. Obviously, any other algorithm that cannot perform sufficiently better than a uniform random one is not worth investigating further.

In most large parallel distributed systems, the transmission time of messages is proportional to the actual communication distance traveled. In such systems, it is advantageous to place objects which communicate frequently closer to each other. One measure of determining which objects will communicate frequently is the parent/child relationship. If an object creates another object, then chances are that the parent object will reference the child object. This assumption is the motivation for using object creation (and, therefore, resource allocation) for determining locality of reference.

The particulars of the object-oriented resource allocation problem that we have studied reduce any potential benefit that can be gained through decreasing the message communication distance (and hence the latency) between objects. This effect is caused by the relatively low ratio of computation to communication within method invocations. As a result, the message queues on the processors tend to be long enough to hide the effects of message latency.

Locality of reference is only expected to be important if the application includes synchronous communication operations, or if the amount of communication is neither too high (to fill up the processor queues) nor too low (so that increasing the performance of the communication would not significantly alter the execution time of the entire application). However, we did investigate a simple scheme that can be used in place of uniform random allocation which does reduce message distances. This scheme uses a non-uniform random distribution biased towards nearby destinations.

2.2.2 Ideal

At the opposite end of the scale, we include an algorithm in our tests that exploits knowledge about future execution behavior of the program that no realistic allocation algorithm can possibly use. This ideal algorithm uses a metric that assigns to each processor a weight based on the processor's exact future load from the objects currently allocated on it. The metric does not need to take into account any synchronization-induced blocking that may take place during the future computations on these objects, since the processor is available for computation on other objects while blocking. Once all processors are assigned a weight, the ideal algorithm always picks the processor with the least load for the destination of the allocation.

The ideal algorithm also uses the unrealistic protocol in which the source processor of the allocation updates the destination processor's state atomically, increasing the future load metric by the amount of computation required by the allocated object. Furthermore, the source processor ensures that the destination does have sufficient memory to hold the object prior to allocation; otherwise the source processor chooses the next best destination. The resulting algorithm, as mentioned previously, is not optimal, since it does not take into account future object creations when making its decision. However, the performance of this ideal algorithm is an upper bound on the performance of all practical algorithms, which cannot possibly use information about future object creations either.

2.3 TEST CASE ALGORITHMS

There are literally hundreds of combinations of load metrics, choice method, and protocol. Furthermore, since our simulation takes hours to run for a test of a single combination, all combinations were not tested. Since the primary goal of our study is to find a suitable algorithm that performs well (better than random and close to ideal), we used an approach that narrows the search to algorithms which appear most promising. We admit the possibility that we may have missed a combination with advantages greater than those tested. We include as an appendix five charts and their descriptions that summarize some of our simulation results.

SECTION 3

SIMULATION TECHNIQUE

In order to determine the performance of our baseline and test cases, we constructed a discrete event simulation of a 1024-processor multicomputer with a four nearest neighbor mesh topology. The simulation was coded in C++ using our own discrete event simulation library, MOOSE (MITRE Object-Oriented Simulation Executive). MOOSE adds several classes (primarily the *process* class) which facilitate the development of process model, discrete event, object-oriented, thread-based simulations. For a more complete description of MOOSE, see [3].

Within the simulation, each processor is represented as having an input queue for messages, a table containing objects, and a CPU resource. All messages received by a processor are put on the same input queue, which is FIFO. Messages on the input queue get serviced one at a time by the single processor resource.

When a message is sent, the transmission time is calculated from the distance between the source and destination (without taking into account network traffic). A message reception process is then scheduled for the destination processor object at the calculated receipt time. The reception process is responsible for queuing itself on the destination processor's input queue, obtaining the destination processor's CPU resource, and finally simulating the actual execution of the message.

Operating system messages, such as allocation requests, execute immediately once they acquire the destination processor's CPU resource. Application messages are passed on to the target object, where they are queued once again. The application object queues, rather than being simply FIFO, pass messages in the order corresponding to the sequential execution order of messages recorded for that object. When an application message arrives that can execute without blocking (without waiting for an "earlier" message to arrive), the corresponding reception process keeps control of the CPU resource during its execution. If the next message in the sequential execution order is already queued on the object, the executing message will pass the CPU resource to it immediately on completion.

Each application message process consists of a loop. For each iteration of the loop, the process consumes some CPU time and either sends an application message to some other object or requests that an object be created. The simulation time required for sending messages and requesting object creations is parameterizable.

The reception processes of operating system messages, such as object creation requests and refusals, also consume simulation time during their execution.

Garbage collection is simulated by removing each object from its host processor when the object finishes processing its final message.

The simulation is started by placing an initial object on some processor in the mesh, usually in the center of the mesh. A startup message is scheduled for the initial object at simulation time 0. The initial object then begins the simulation by executing the startup message's reception process.

To drive our resource allocation simulations, we developed a tool for recording the characteristics of object-oriented programs. With this tool, we are able to record specific objects dynamically created by a program, and the number and sequence of messages sent and received by each object. The application programs recorded to drive the simulation, however, are graph construction and traversal programs devised specifically to generate very large parameterizable program traces with large numbers of objects and messages. Since we do not have any large object-oriented applications available, the approach of generating synthetic applications is necessary for our tests.

Using the program trace constructed during the application program's sequential execution, we are able to drive the simulation deterministically. The synthetic applications we used for our tests contained about 20,000 objects and 250,000 application messages. Depending on the resource allocation algorithm being tested, the real time needed for each simulation run ranged from three hours to nearly five hours on a SUN SPARCStation 1. The simulations each consumed between forty and fifty megabytes of memory. The SPARCStation used for testing contained 64 megabytes of RAM, so paging overhead was low.

The granularity of computation between message sends and the size of the applications objects are exponentially distributed with maximum cut-off values. Because of the large variances in exponential distributions, the scenarios generated are likely to make load-balancing more difficult than would real applications. This augments the difference between the ideal and random baseline cases, and provides a greater resolution in the test results.

SECTION 4

RESULTS

Perhaps our most ubiquitous result is that it is quite difficult to construct an allocation algorithm that can perform much better than the baseline random algorithm. Even the ideal baseline algorithm only performs about 25 percent better than random. Using a round-robin allocation policy in place of a uniform random one offers a slight improvement in some of our tests. This improvement is probably due to the tendency of a round-robin algorithm to balance allocations from the perspective of each source processor, thus providing a lower standard deviation of load per processor than that provided by a random distribution. However, in most cases, the improvement noticed is too small to consider more than a statistical aberration.

Another surprising result is the tendency for many algorithms with a specific choice method and protocol to actually decrease in performance as the amount of knowledge incorporated in the allocation decision increases. These algorithms are prone to a phenomenon, *piling on*, in which a small number of destination processors are inundated by resource allocation requests. Piling on happens when simultaneous or nearly simultaneous allocation requests from different processors occur, and only a small number of destination processors are considered favorable destinations for allocation. The choice methods responsible for piling on are those which narrow the allocation algorithm's focus to a small number of possible destinations that have low loads. These methods simulate the greater use of load information in an allocation algorithm. One would expect that this increased use of information would result in greater performance. However, because fewer processors qualify for each allocation, requests pile up on the message queues of the qualifying processors, creating a bottleneck. Once the favorable processors accept the first few allocations, they become less favorable places for the remaining allocations. If these algorithms use the source measurement protocol, the remaining allocations will not be refused by the destination (unless the destination runs out of space). When the number of nearly simultaneous allocations is sufficiently high, the chosen destinations will fill quickly to capacity, severely unbalancing processor load in the system. Regardless of what metric is used, the resulting simulation runs are three to four times slower than the random baseline.

The relative performance of the load metrics in all of the tests was much less surprising. The future load metric consistently outperforms all of the others. Furthermore, all of the other metrics resulted in nearly equivalent performance. The number-of-objects metric was the best in this remaining group, but this result is probably due to the independence of object size and object load resulting from our pseudo-application generation scheme. For applications where object size and load are correlated, the memory-use metric would probably result in greater performance. However, because of the small performance differences between the metrics tested, we did not construct pseudo-applications to test this hypothesis. We instead concentrated on finding solutions to the piling-on effect.

Once we spotted the piling-on effect and determined its cause, we decided on three different possible fixes. The first response was to incorporate atomic test-and-set as a possible protocol. The atomic protocol has the source processor testing the possible destinations, choosing the actual destination, and updating the actual destination's load accordingly without elapsing simulation time. As a result, other source processors doing simultaneous allocations will see the chosen destination with its updated load. The atomic protocol is obviously impractical in an asynchronous system, but the resulting algorithms can serve as best case tests for each of the different load metrics.

The second answer to the piling-on effect increases the size of the initial set of possible destinations produced by the processor choice mechanism. The set would be formed by incorporating processors that have loads lower than the average system load plus or minus some number of standard deviations. The source processor would then choose the actual destination from this set either randomly or round-robin.

The most practical response to piling on is to have the destination processor do the work of determining if it is an appropriate allocation site. In such schemes, the source processor would choose the destination either randomly or round-robin. The destination, upon receipt of the allocation request, would compare its load to the average system load. Again, the allocation request would be accepted if the destination processor's load is lower than the average load plus or minus some number of standard deviations. If the allocation is refused, the destination processor would forward the request to some other processor chosen either randomly or round-robin.

Simulation results of the above approaches indicate that all three consistently perform better than the random baseline, regardless of the metric used. Also, for each metric, the relative performance difference among the three methods never varied more than 15 percent. Finally, the variation in performance with either the second or third approach as the number of standard deviations above the average was varied between 0 and 2 remained below 10 percent, despite the large differences in the number of refused allocations in the third approach. This is quite surprising, considering that the variation (between 0 and 2 standard deviations) in the number of processors allowed as destinations is 47 percent (if the distribution is normal). We did notice, however, that the third approach degrades when the number-of-objects metric is used and the number of standard deviations from the average is 0. This is due to the fact that there is no variation in the number-of-objects metric per object allocated (it always increases by one), so processors fill up synchronously: every processor gets one object before any is allowed to get a second and so on. This generates an enormous number of allocation refusals as the number of acceptable destinations decreases from 1024 to 1 in each cycle. (We suspect that any way of estimating the average number of objects that allows overestimates would alleviate this problem.)

The small amount of variation in the performance of the third approach (where the destination processor does the testing) as the number of standard deviations from the average

changes (with the above caveat for the number-of-objects metric) leads us to believe that the algorithms can perform well with very imprecise information about the global load average. Cases where each processor's view of the load is out of date, or only a statistical sample of part of the mesh (such as the local neighborhood), should not suffer a performance degradation beyond the 10 percent we observed. Our target operating system already incorporates a periodic global collect and broadcast algorithm (see the discussion of *global virtual time* in [17]); the messages used in this algorithm can be easily modified to carry and distribute processor load information.

SECTION 5

CONCLUSIONS

We have investigated several resource allocation algorithms. A resource allocation algorithm for our purposes takes care of assigning dynamically created objects to processors, taking into account the current state of the system. We classify resource algorithms by their load metrics, choice methods, and protocols.

Examples of processor load metrics are the number of objects on a processor, the amount of memory used, the length of the message queue, the average wait time on a message queue, and remaining execution time of objects on the processor. In addition, each object to be allocated must fit in the destination processor's local memory.

Processor choice methods involve the determination of the destination once the loads for all processors are known (in some fashion, such as the average). The investigated choice methods include choosing the best processor based on the load, choosing one of the X percent best processors, for some X, and choosing a processor with a load that is not greater than the average plus or minus X standard deviations, for some X.

The protocol portion of an allocation algorithm involves the determination of which of the two processors involved does the measurement of the loads and when information about the loads is updated.

Our tests failed to demonstrate a *significant* performance advantage for any of the plausible processor load metrics, given the specific performance characteristics of the particular applications used to obtain our results. The one implausible metric, future processor load, consistently outperforms the rest, but only by about 10 percent.

The allocation protocol plays a significant role: preventing an effect we refer to as piling on. Piling on results when a large number of nearly simultaneous allocations all choose from the same small set of acceptable destinations. The resulting bottlenecks can reduce performance considerably.

The most promising algorithms have the source processor attempt an allocation based on some *choice method* independent of processor load. When the destination processor of the allocation receives the request, it determines if it is a suitable location based on a comparison between its local load and an estimate of the average processor load in the system. If the destination is not suitable, it forwards the request to some other processor, which then decides if it is an acceptable location, and so on. Our observations indicate that this technique is fairly insensitive to errors in the determination of the global load average due to out-of-date or approximate data.

LIST OF REFERENCES

[1] Bensley, E. H., T. J. Brando, and M. J. Prelle, September 1988, "An Execution Model for Distributed Object-Oriented Computation," *Proc. of the Third Annual Conference on Object Oriented Programming Systems, Languages, and Applications*, pp. 316-322.

[2] Bensley, E. H., T. J. Brando, J. C. Fohlin, M. J. Prelle, and A. M. Wollrath, January 1989, "Distributed Object Oriented Programming," Technical Report MTR-10531, The MITRE Corporation.

[3] Bensley, E. H., V. T. Giddings, J. I. Leivent, and R. J. Watro, 1991, "A Performance-based Comparison of Object-oriented Simulation Tools," submitted to the West Coast Simulation Conference.

[4] Bolosky, W. J., R. P. Fitzgerald, and M. L. Scott, December 1989, "Simple But Effective Techniques for NUMA Memory Management," *Proc. of the Twelfth ACM Symposium on Operating Systems Principles*, pp. 19-31.

[5] Cox, A. L. and R. J. Fowler, December 1989, "Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM," *Proc. of the Twelfth ACM Symposium on Operating Systems Principles*, pp. 32-44.

[6] Eager, D. L., E. D. Lazowska, and J. Zahorjan, May 1986, "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Transactions on Software Engineering*, Vol. 12, No. 5, pp. 662-675.

[7] Goldberg, A. and D. Robinson, 1983, *Smalltalk-80: The Language and its Implementation*, Reading, MA: Addison-Wesley.

[8] Hac, A., February 1989, "Load Balancing in Distributed Systems: A Summary," *Performance Evaluation Review*, Vol. 16, No. 2, pp. 17-19.

[9] Hailperin, M., September 1988, "Load Balancing for Massively-Parallel Soft-Real-Time Systems," Technical Report STAN-CS-88-1222, Department of Computer Science, Stanford University.

[10] Iqbal, M. A., J. H. Saltz, and S. H. Bokhari, August 1986, "A Comparative Analysis of Static and Dynamic Load Balancing Strategies," *Proc. of the 1986 International Conference on Parallel Processing*, pp. 1040-1047.

[11] Jefferson, D., et al., November 1987, "Distributed Simulation and the Time Warp Operating System," *Proc. of the Eleventh ACM Symposium on Operating Systems Principles*, pp. 77-93.

[12] Jul, E., H. Levy, N. Hutchinson, and A. Black, February 1988, "Fine-Grained Mobility in the Emerald System," *ACM Transactions on Computer Systems*, Vol. 6, No. 1, pp. 109-133.

[13] Mirchandaney, R., D. Towsley, and J. A. Stankovic, November 1989, "Analysis of the Effects of Delays on Load Sharing," *IEEE Transactions on Computers*, Vol. 38, No. 11, pp. 1513-1525.

[14] Ngai, T-F., October 1986, "Dynamic Resource Allocation in a Hierarchical Multiprocessor System, A Preliminary Study," Technical Report CSL-TR-86-310, Computer Systems Laboratory, Stanford University.

[15] Nicol, D. M. and P. F. Reynolds, February 1990, "Optimal Dynamic Remapping of Data Parallel Computations," *IEEE Transactions on Computers*, Vol. 39, No. 2, pp. 206-219.

[16] Nicol, D. M. and J. H. Saltz, September 1988, "Dynamic Remapping of Parallel Computations with Varying Resource Demands," *IEEE Transactions on Computers*, Vol. 37, No. 9, pp. 1073-1087.

[17] Prelle, M. J., T. J. Brando, E. H. Bensley, J. I. Leivent, R. J. Watro, and A. M. Wollrath, December 1990, "Distributed Object-Oriented Programming, FY90 Final Report," Technical Report MTR-11058, The MITRE Corporation.

[18] Scheurich, C. and M. Dubois, August 1989, "Dynamic Page Migration in Multiprocessors with Distributed Global Memory," *IEEE Transactions on Computers*, Vol. 38, No. 8, pp. 1154-1163.

[19] Tanenbaum, A. S. and R. van Renesse, December 1985, "Distributed Operating Systems," *ACM Computing Surveys*, Vol. 17, No. 4, pp. 419-470.

[20] Tantawi , A. N. and D. Towsley, April 1985, "Optimal Static Load Balancing in Distributed Computer Systems," *Journal of the ACM*, Vol. 32, No. 2, pp. 445-465.

APPENDIX

SAMPLE SIMULATION CHARTS

The execution time shown in the figures in this appendix is the total execution time of a recorded application program with 20,000 objects and 250,000 application messages whose execution was simulated on a mesh of 1,024 processors. The first two columns in each figure show the execution times of the two baseline algorithms. Figure 1 shows that the round-robin choice method can perform slightly better than the random baseline. The rightmost column is an example of piling on. The algorithm is just the ideal baseline with one change: the destination's load is not updated atomically, but after the allocation is accepted.

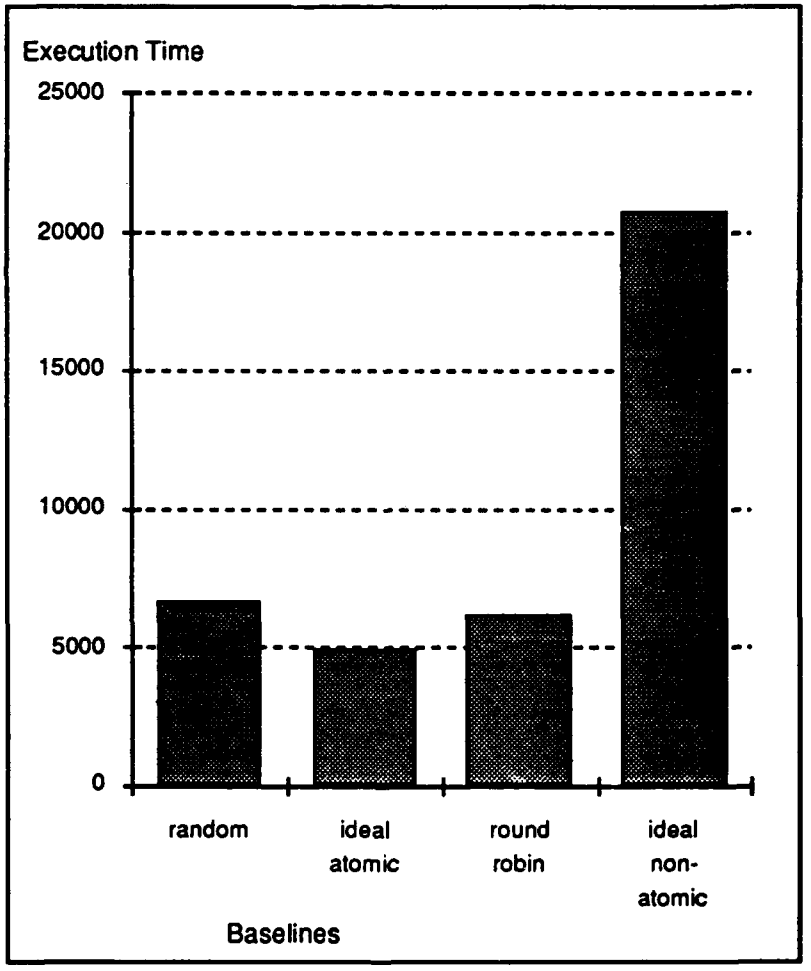


Figure 1. Round Robin and Piling On

Figure 2 shows that piling on can be reduced significantly by increasing the number of acceptable processors in the choice method. Each of the double columns shows the performance of an algorithm with the future-load metric (the same metric used in the ideal atomic baseline and ideal non-atomic piling-on examples). The percentage labels indicate the number of processors considered acceptable by the choice method (as a percentage of the entire mesh). The first column in each pair uses the atomic update protocol, the second column does not.

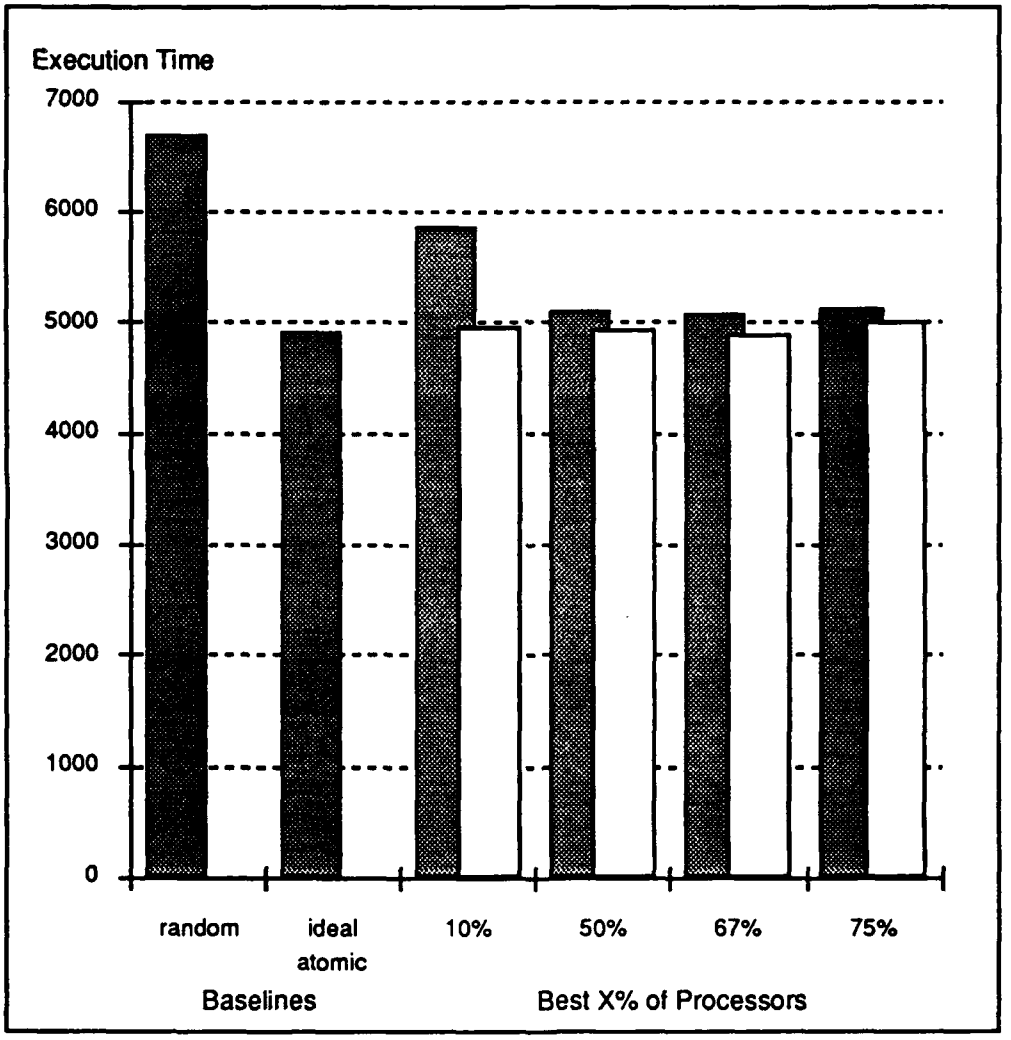


Figure 2. Atomic vs. Non-Atomic Protocol

Figure 3 compares three algorithms using the future-load metric and three different choice mechanisms. The first column of each triple results from using the destination-measure protocol (where the destination processor decides if it is an acceptable host for the allocation), with the source protocol choosing potential destinations randomly. The second column results from a similar algorithm that has the source processor choose potential destinations round-robin instead of random. The third column (for which there is only data in the 0- and 1-standard deviation range) results from having the source processor do all of the work, including updating the destination's load atomically. The graph shows that the practical choice mechanisms (the first two in each triple) perform nearly as well as one that uses atomic updating.

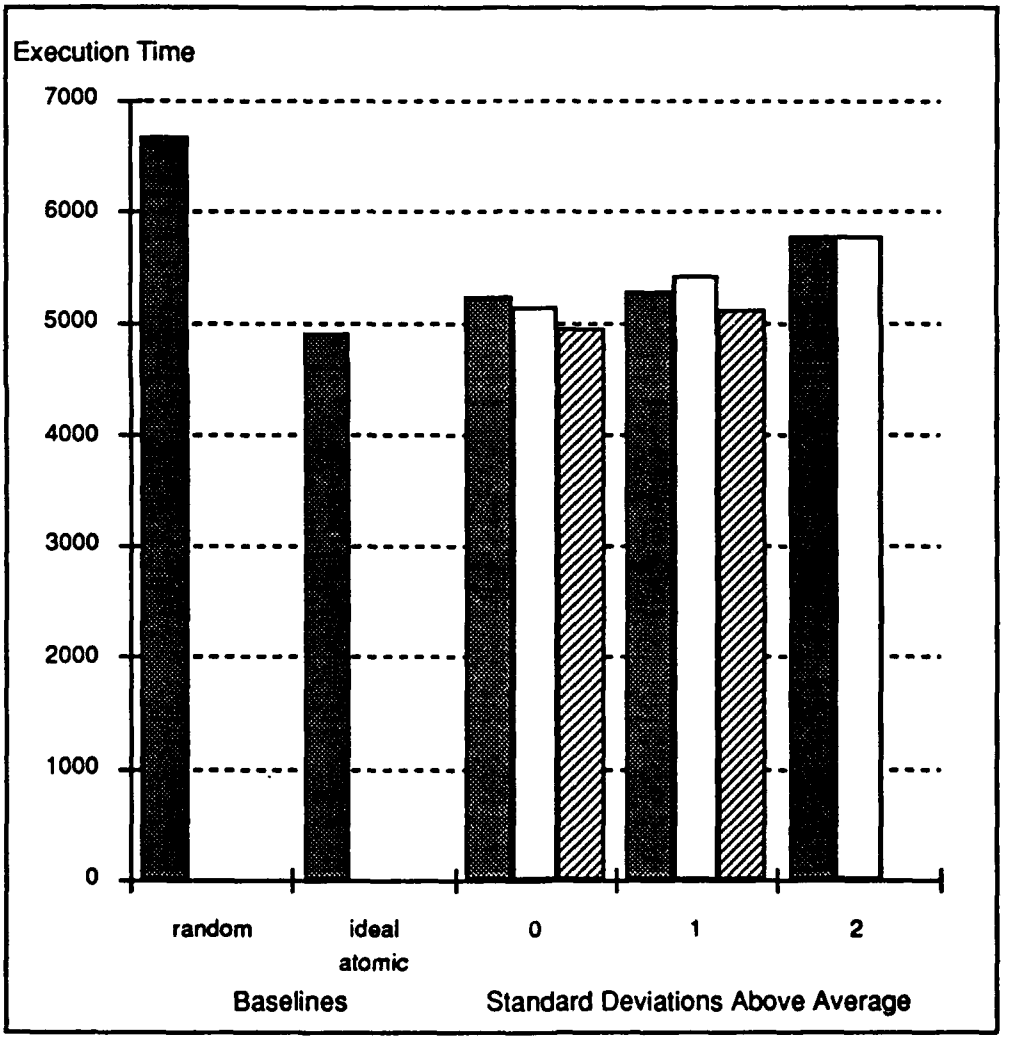


Figure 3. Future-Load Metric

Figure 4 illustrates the performance of the number-of-objects metric. The 1- and 2-standard deviation column pairs result from the practical destination-measuring choice method. The first column in each pair results from having the source processor choose potential destinations randomly; the second column is round-robin. The 0-standard deviation column results from a protocol that has the source processor do all of the work. Data for 0 standard deviations above the average for the destination-measuring case is not included, because of the anomaly mentioned in the discussion of results.

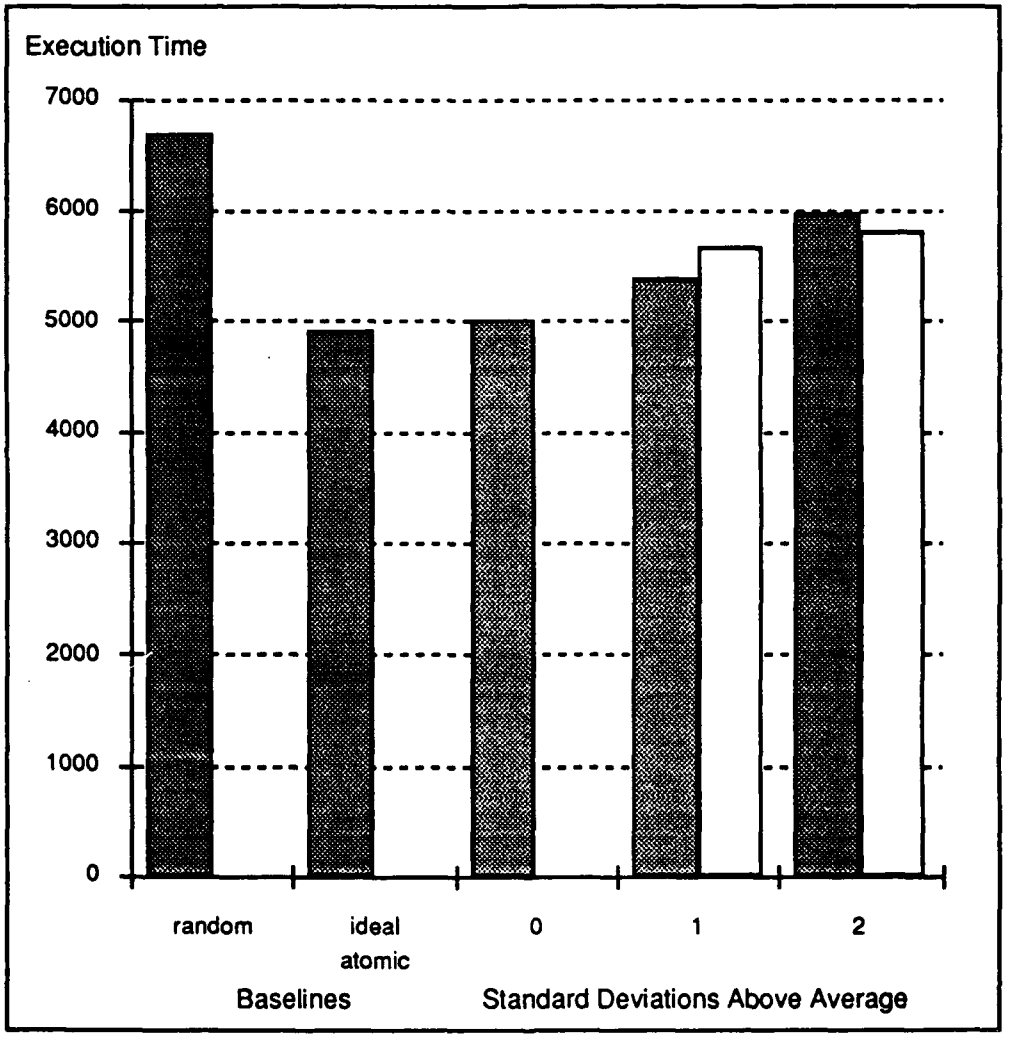


Figure 4. Number-of-Objects Metric

Figure 5 illustrates the performance of the memory-use metric. The first three columns to the right of the baselines result from the destination-measuring choice method, with the source choosing potential destinations randomly. The last column shows the performance of the impractical source-measuring protocol that updates the destination's load atomically. Again, the performance advantage of the impractical case is small.

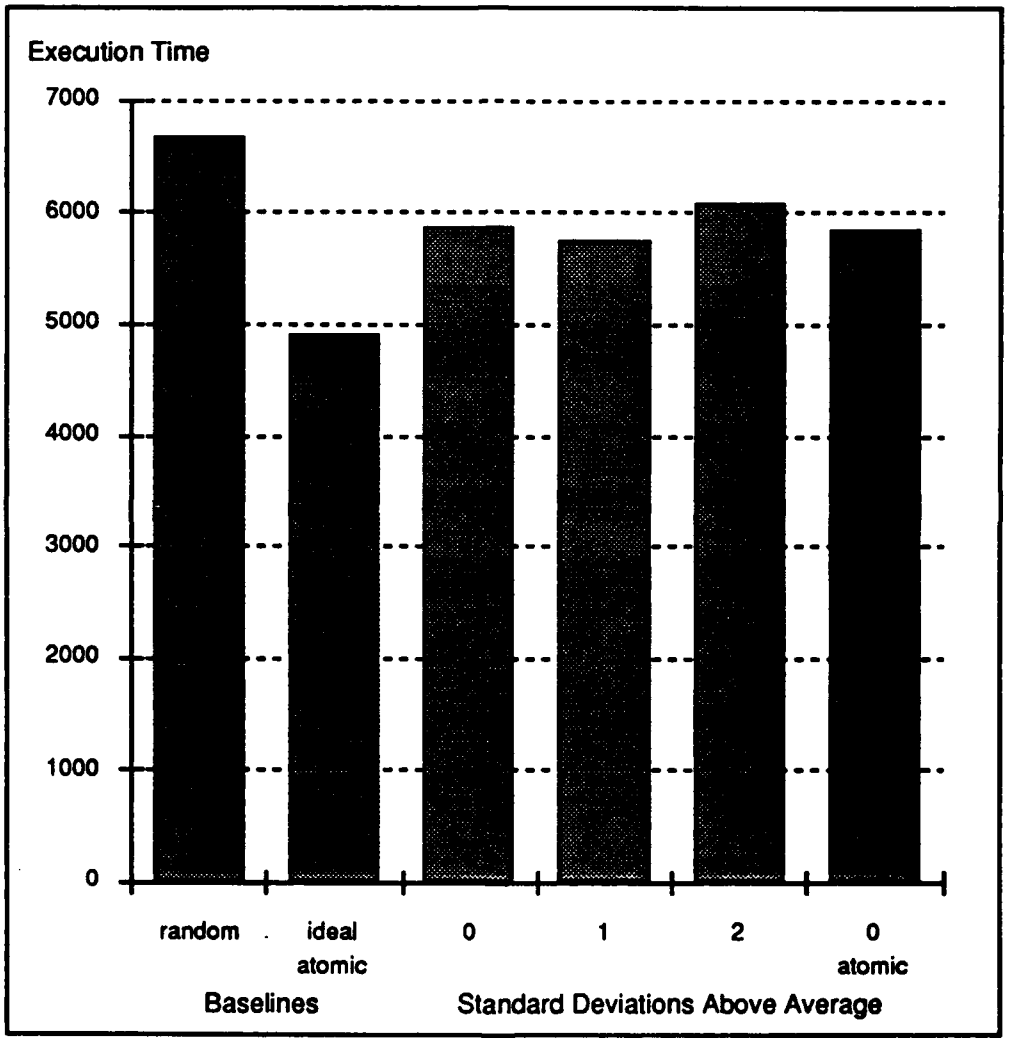


Figure 5. Memory-Use Metric